Day 9
Advanced Linux Programming
Part II

Joel Isaacson
Ascender Technologies Ltd.
http://ascender.com

Today's Lectures

- Today we will continue to examine the Linux API
  - Zero Copy Networking
  - Advanced File I/O
  - Process Management
  - File Management
  - Memory Management

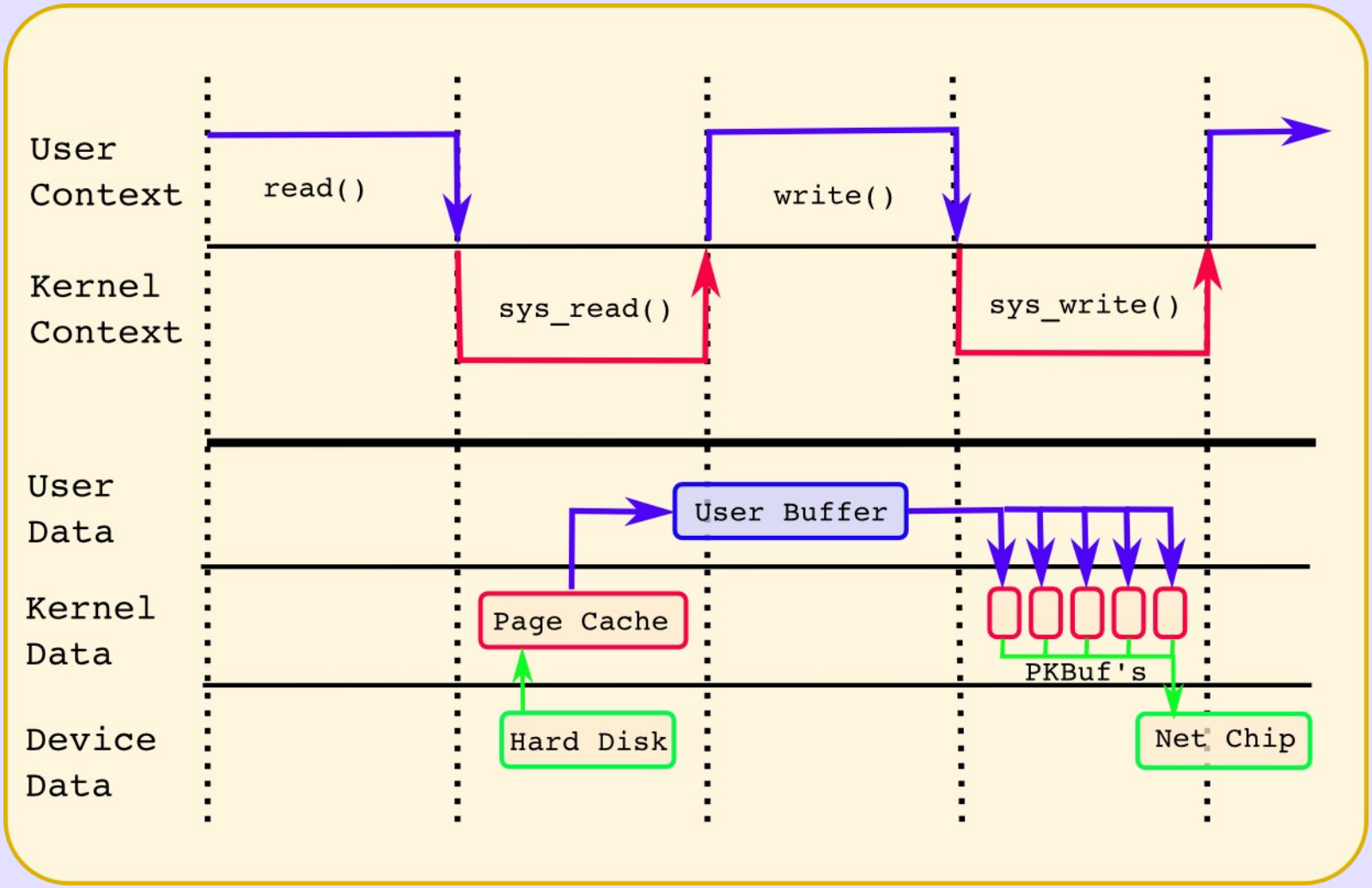
Zero Copy

- Sometimes we can conduct our business without actually transferring data to userspace.
- The classic example of an application that can benefit from “zero copy” is an Ftp, Web or Video server.

Two Copy

- Typically in these applications we read data from a file on disk and write it out on a Tcp/Ip socket.
- The naive read(file,...)/write(sock,..) code would:
  - Transfer the data to the kernel page cache
  - Read the data to user space
  - Write the data to kernel space
  - Copy the data to the network device

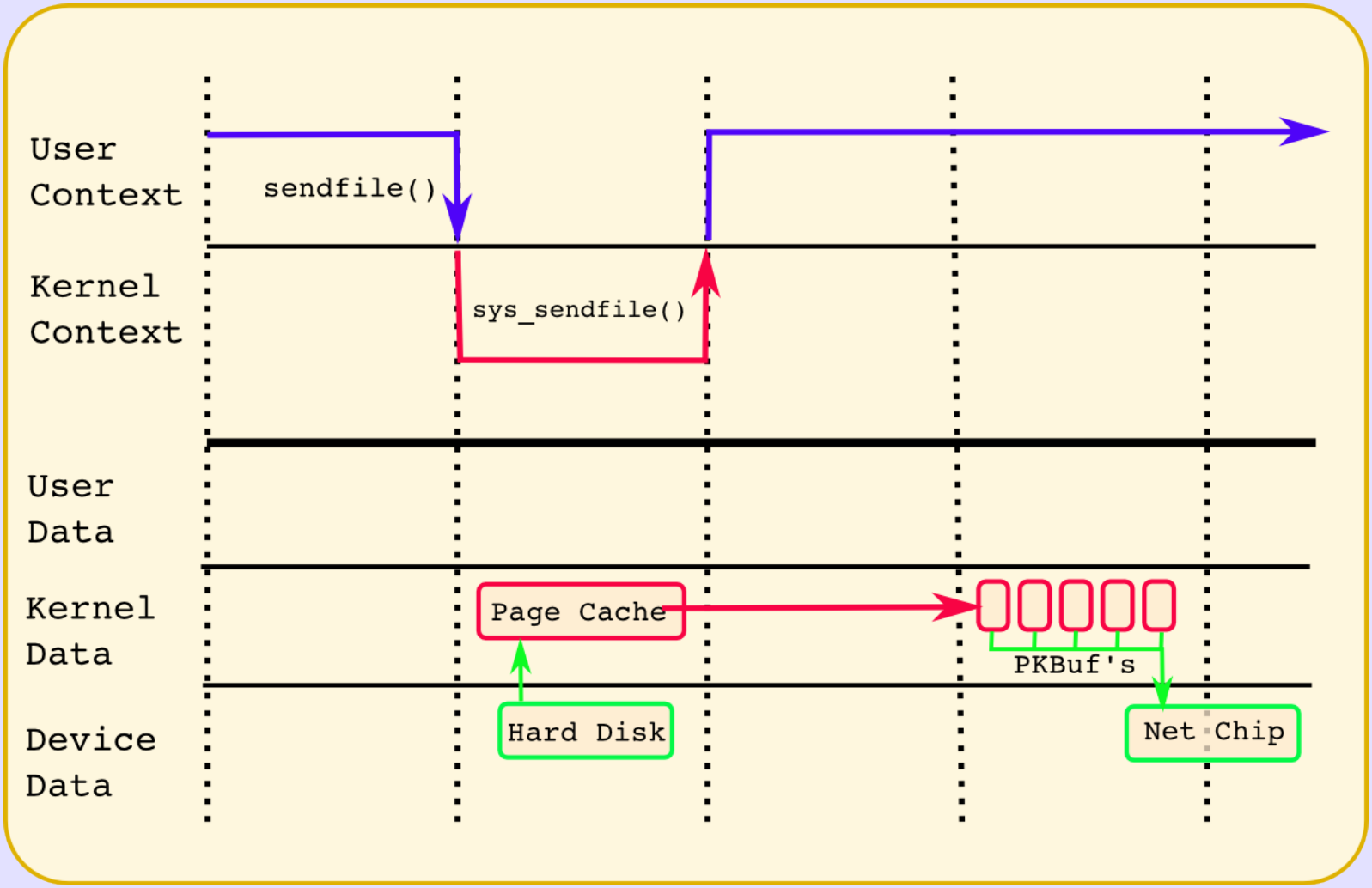
Two Copy


User Context
read()
write()
Kernel Context
sys_read()
sys_write()
User Data
User Buffer
Kernel Data
Page Cache
PKBuf's
Device Data
Hard Disk
Net Chip

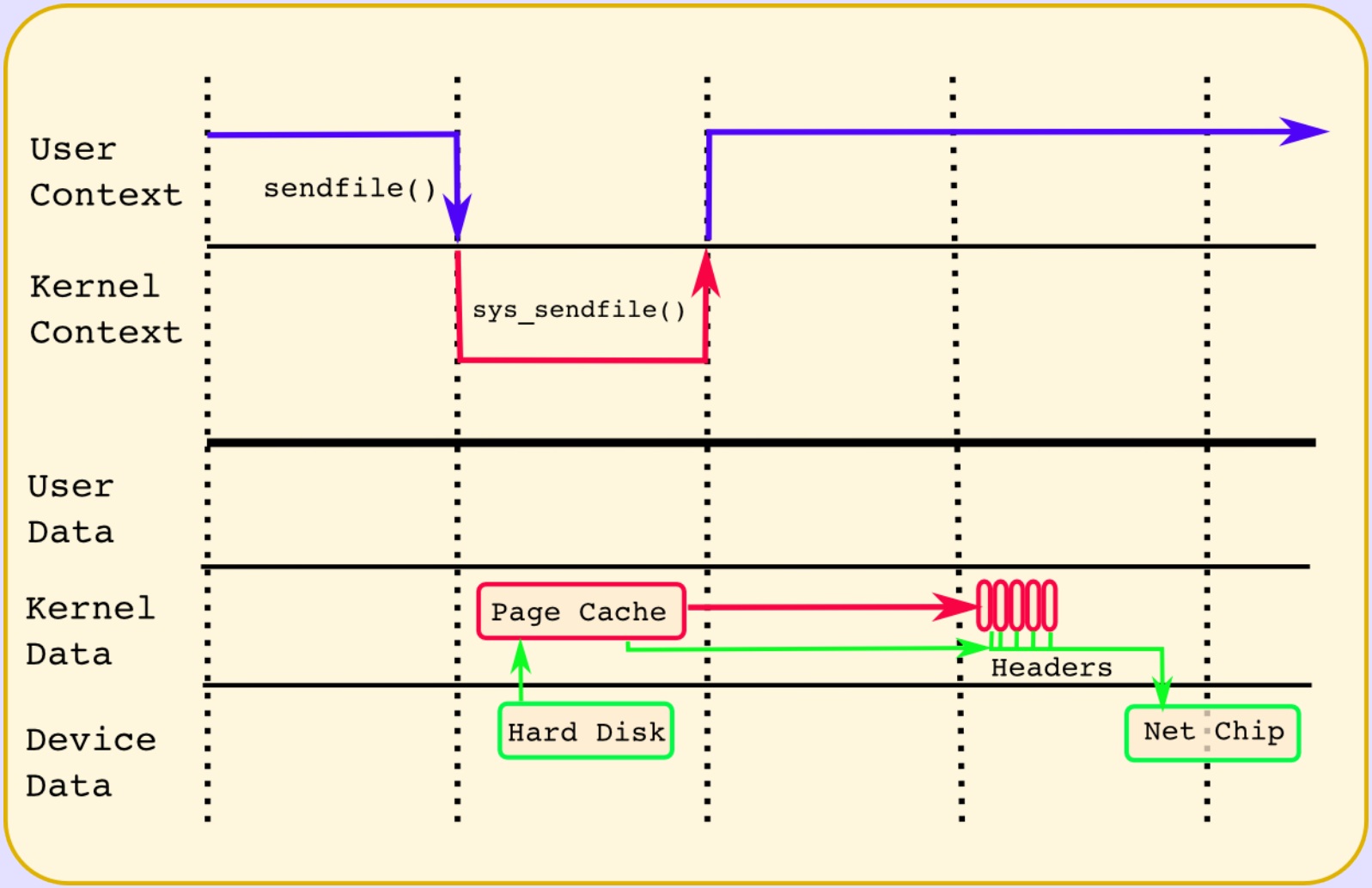
Zero Copy - Sendfile

- In kernel version 2.1, the sendfile system call was introduced to simplify the transmission of data over the network and between two local files. Introduction of sendfile not only reduces data copying, it also reduces context switches.
- Use it like this:
    - sendfile(socket, file, len);

One Copy


User Context
sendfile()
Kernel Context
sys_sendfile()
User Data
Kernel Data
Page Cache
PKBuf's
Device Data
Hard Disk
Net Chip

Zero Copy - Better



Splice/Vmsplice/Tee

- splice() is a system call that copies data between a file handle and a pipe, or between a pipe and user space.
- It does so without actually copying the data, in contrast to other data copying techniques, thereby improving I/O performance.

Splice/Vmsplice/Tee

- The crucial service offered by splice is that one can move data from one file descriptor to another without incurring any copies from user space into kernel space, which is usually required to enforce system security and also to keep a simple and elegant interface for processes to read and write to files.

Splice/Vmsplice/Tee

- The key insight that splice exploits is that a pipe buffer is effectively implemented as an in-kernel memory buffer that is opaque to the user space process.

Splice/Vmsplice/Tee

- This means that the user process can splice the contents of a source file into this pipe buffer, without ever copying the contents of the source file into memory, then into kernel space, then splice the contents of the pipe buffer into the destination file, again without incurring any copies

Advise For File I/O
posix_fadvise()

```
posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- Programs can use to announce an intention to access file data in a specific pattern in the future, thus allowing the kernel to perform appropriate optimizations.

Advise For File I/O
posix_fadvise()

```
posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- The advice applies to a region starting at offset and extending for len bytes (or until the end of the file if len is 0) within the file referred to by fd.
- The advice is not binding; it merely constitutes an expectation on behalf of the application.

posix_fadvise()

```
posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- POSIX_FADV_NORMAL - Indicates that the application has no advice to give about its access pattern for the specified data.
- POSIX_FADV_SEQUENTIAL - The application expects to access the specified data sequentially (with lower offsets read before higher ones).

posix_fadvise()

```
posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- POSIX_FADV_RANDOM - The specified data will be accessed in random order.
- POSIX_FADV_NOREUSE - The specified data will be accessed only once.

posix_fadvise()

```
posix_fadvise(int fd, off_t offset, off_t len, int advice)
```

- POSIX_FADV_WILLNEED - The specified data will be accessed in the near future.
- POSIX_FADV_DONTNEED - The specified data will not be accessed in the near future.

I/O Schedulers (Robert Love)

- To understand the role of an I/O scheduler, let's go over some background information and then look at how a system behaves without an I/O scheduler.
- Hard disks address their data using the familiar geometry-based addressing of cylinders, heads and sectors.

I/O Schedulers

- A hard drive is composed of multiple platters, each consisting of a single disk, spindle and read/write head.
- Each platter is divided further into circular ring-like tracks, similar to a CD or record.
- Finally, each track is composed of some integer number of sectors.

I/O Schedulers

- Modern hard disks do not force computers to communicate with them in terms of cylinders, heads and sectors.
- Instead, modern hard drives map a unique block number over each cylinder/head/sector triplet.
- The unique number identifies a specific cylinder/head/sector value.

I/O Schedulers

- Moving the disk head is called seeking, and it is one of the most expensive operations in a computer.
- The seek time on modern hard drives is measured in the tens of milliseconds.
- This is one reason why defragmented files are a good thing.

I/O Schedulers

- Unfortunately, it does not matter if the files are defragmented because the system is generating I/O requests for multiple files, all over the disk.
- The net effect is that the disk head is made to jump around the disk.

I/O Schedulers

- This is where the I/O scheduler comes in.
- The I/O scheduler schedules the pending I/O requests in order to minimize the time spent moving the disk head.
- This, in turn, minimizes disk seek time and maximizes hard disk throughput.

I/O Schedulers

- This magic is accomplished through two main actions, sorting and merging.
  - Sorting: The I/O scheduler keeps the list of pending I/O requests sorted by block number.
  - Merging: Merging occurs when an I/O request is issued to an identical or adjacent region of the disk. This minimizes the number of outstanding requests.

The Linus Elevator

- The I/O scheduler found in the 2.4 Linux kernel is named the Linus Elevator.
- I/O schedulers often are called elevator algorithms, because they tackle a problem similar to that of keeping an elevator moving smoothly in a large building.

The Linus Elevator

- The Linus Elevator functions almost exactly like the classic I/O scheduler described above.
- Unfortunately, in the I/O scheduler's quest to maximize global I/O throughput, a trade-off was made: local fairness—in particular, request latency—can go easily out the window.

The Linus Elevator Read/Write Unfairness

- Read requests generally are synchronous (blocking).
- When an application issues a request to read some data, it typically blocks and waits until the kernel returns the data.

The Linus Elevator Read/Write Unfairness

- When an application issues a write, the kernel copies the data and metadata into the kernel, prepares a buffer to hold the data and returns to the application.
- The application does not really care or even know when the data actually hits the disk.

The Linus Elevator Read/Write Unfairness

- Because writes are asynchronous, writes tend to stream.
- That is, it is common for a large writeback of a lot of data to occur.
- The application dumps write requests on the system and hard drive as fast as it is scheduled.

The Deadline I/O Scheduler

- The Deadline I/O Scheduler was introduced to solve the starvation issue surrounding the 2.4 I/O scheduler and traditional elevator algorithms in general.
- It adds two additional queues
  - the read FIFO queue
  - the write FIFO queue.

The Deadline I/O Scheduler

- The Deadline I/O Scheduler keeps the items in each of these queues sorted by submission time.
- Each FIFO queue is assigned an expiration value.
  - The read FIFO queue has an expiration time of 500 milliseconds.
  - The write FIFO queue has an expiration time of five seconds.

The Deadline I/O Scheduler

- When the item at the head of one of the FIFO queues, hits its expiration value, the I/O scheduler stops dispatching I/O requests from the standard queue.
- Instead, it services the I/O request at the head of the FIFO queue, plus a couple extra for good measure.

The Deadline I/O Scheduler

- In this manner, the Deadline I/O Scheduler can enforce a soft deadline on I/O requests.
- Thus, the Deadline I/O Scheduler continues to provide good global throughput without starving any one request for an unacceptably long time.

Anticipatory I/O Scheduler

- Because reads are issued in dependent chunks, the application issues the next read only when the previous is returned.
- But by the time the application receives the read data, is scheduled to run and submits the next read, the I/O scheduler has moved on and begun servicing some other requests.

Anticipatory I/O Scheduler

- When a read request is submitted, the Anticipatory I/O Scheduler services it within its deadline, as usual.
- Unlike the Deadline I/O Scheduler, however, the Anticipatory I/O Scheduler then sits and waits, doing nothing, for up to six milliseconds.

Anticipatory I/O Scheduler

- Chances are good that the application will issue another read to the same part of the filesystem during those six milliseconds.
- If so, that request is serviced immediately, and the Anticipatory I/O Scheduler waits some more.

CFQ I/O Scheduler

- The Completely Fair Queueing (CFQ) i/o scheduler tackles the unfairness problem by maintaining separate i/o queues for each processes and switching between i/o queues in a round robin fashion.

Selecting a I/O Scheduler

- It can be selected globally in the boot line:

elevator={"anticipatory" | "cfq" | "deadline" | "noop"}

- It can be selected per mounted file system:

echo qfq > /sys/block/sda/queue/scheduler

- The CFQ scheduler can set i/o priorities per process via the “ionice” program.

AIO – Asynchronous I/O

- AIO enables even a single application thread to overlap I/O operations with other processing. It provides two interfaces:
- One for submitting I/O requests in one system call (io_submit()).
- A separate interface (io_getevents()) to reap completed I/O operations.

AIO – Asynchronous I/O

- Linux AIO is limited to files that are opened with the O_DIRECT mode.
- This makes Linux AIO almost useless.

Process Management

- Each process has a unique process id.
- New processes are children of the processes that forked them.
- The process id is gotten from getpid().
- The parent process id is gotten fron getppid();

Running a New Process

- Fork() (or clone()) is used to start a new process. Copy-on-write is used to keep the fork() efficient.
- Execve() is used to change the executable code being run.

Terminating a Process

- A process can be terminated by encountering an error (e.g. SIGSEGV).
- A process can be terminated by the exit() call.
- Programs that “fall off the end” of the main routine will execute a default exit().

Waiting

- Someone has to wait after a process exits to reap the process status.
- Until a terminated process is “waited” for it remains in the system having the status of “zombie”.
- When the process that started the zombie doesn't reap it properly it's grandparent will “reparent” it and then reap the exit value.

Scheduling

- Linux is a multitasking operating system, having many processes resident in memory at one time.
- Each processes has the illusion that it has a CPU of it own.
- Deciding which processes are to be run is the job of the scheduler.

Scheduling

- The scheduler is the kernel component that selects among processes that are runnable.
- The scheduler effectively provides a time-multiplexed CPU to many independent processes.

Scheduling

- Multitasking operating systems come in to varieties:
  - cooperative multiprocessing and
  - preemptive multiprocessing.
- Cooperative multiprocessing (like Windows 3.1) only schedule processes after they yield the CPU.

Scheduling

- With preemptive multiprocessing processes run for a maximum period of time, called a timeslice (or quantum), and then they are halted to give other runnable processes a chance to run.
- Linux uses a classical preemptive multi-priority queue scheduler with some novel innovations that where present in the first Unix system of the 1970's.

Scheduling

- The thing that differentiates the Linux scheduler from RTOS schedulers is dynamic-priority setting and the way that processes are suspended from rescheduling until all other processes have completed their time slices.

Scheduling

- It is important to note that standard Linux (or standard POSIX in fact) actually has three different schedulers.
- There are two real-time (see 'man sched_setscheduler') schedulers, SCHED_FIFO and SCHED_RR, that take precedence over the the standard Linux scheduler, SCHED_OTHER.

Scheduling

- The scheduler policy is the method by which the dynamic priorities are determined.
- The main rational is to allow good interactive behavior (low latency) and to globally optimise CPU utilization.

Scheduling

- Conceptually it is useful to divide processes into two classes:
  - I/O bound (or interactive) and
  - CPU bound (or compute intensive).
- Of course in general a process may be somewhere between these two definitions and might change its class many times during its execution.

Scheduling

- The general idea is to give interactive processes higher priority and longer timeslices so that intercative users of the system experience low latency.
- A interactive processes is defined as a processes that spends most of its time on a wait queue, in a unrunnable state.

Scheduling

- Compute bound processes are mostly in a runnable state.
- This observation has proven extremely successful and is the basis of all Unix schedulers for the last 35 years.

Scheduling

- For example: if the system has two runnable processes, an editor, and a process to compute the value of π, to 20000 digits.
- We would want to give the editor a higher priority. Otherwise the interactive use will be quite unhappy.

Scheduling

- This naturally happens with the standard Linux scheduler since the text editor is normally spending 99% of the time in a wait queue waiting for keyboard input.

Scheduling

- In addition Unix and Linux allows the user to give a nice value which is added to the dynamic priority in order to give the scheduler information from the user about the relative importance of different processes.

Scheduling

- The nice value effect the amount of time the process will consume in relationship to other processes, but no process will starve for CPU time.

Scheduling – O(1) Algorithm

- During the Linux 2.5.x development period, a new scheduling algorithm was one of the most significant changes to the kernel.
- The Linux 2.4.x scheduler, while widely used, reliable, and in general pretty good, had several very undesirable characteristics.

Scheduling – O(1) Algorithm

- The undesirable characteristics were quite embedded in its design.
- The fact that the Linux 2.4.x scheduling algorithm contained O(n) algorithms was perhaps its greatest flaw, and subsequently the new scheduler's use of only O(1) algorithms was its most welcome improvement.

Scheduling – O(1) Algorithm

- The Linux O(1) scheduler does not contain any algorithms that run in worse than O(1) time.
- That is, every part of the scheduler is guaranteed to execute within a certain constant amount of time regardless of how many tasks are on the system.

Scheduling – O(1) Algorithm

- This allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows.
- There are two key data structures in the Linux O(1) scheduler and its design revolves around them:
  - runqueues and
  - priority arrays.

Synchronization
Posix Semaphores - posix.c

```
#include <semaphore.h>
int main(int argc, char *argv[])
{
  sem_t sem;
  int i;
  sem_init(&sem, 1, 1);
  for(i=0;i<10000000;i++) {
    sem_wait(&sem); // Down
    sem_post(&sem); // Up
  }
}
```

Synchronization
Sys V (ATT) Semaphores

```
#include <linux/sem.h>
struct sembuf op;
int main(int argc, char *argv[])
{
  int semid, i;
  union semun arg;

  semid = semget(1234, 1, 0666|IPC_CREAT);
  arg.val = 1; semctl(semid, 0, SETVAL, arg);
  for(i=0;i<10000000;i++) {
    op.sem_op = -1; semop(semid, &op, 1); // Down
    op.sem_op = 1;  semop(semid, &op, 1); // Up
  }
}
```

Sys V vs Posix Semaphores

- The manual pages for Posix and Sys V semaphores don't seem much different.
- The programs do exactly the same things.
- The Sys V programs run about 24 time slower than the Posix program on Linux.
- The Linux implementation is at least 3 times faster than OpenSolaris.

Sys V vs. Posix Semaphores Linux vs OpenSolaris

	SYS V	POSIX
Linux	17.26, 2.89, 14.38	0.71, 0.71, 0.0
OpenSolaris	51.38, 24.33, 26.98	45.13, 30.74, 14.34

Total time, user time, system time

Sys V vs. Posix Semaphores

- The Sys V implementation is totally kernel based. Each semaphore call takes one system call.
- The Posix implementation is a user/kernel optimized mix based on kernel **futex**'es. Only when a semaphore is contested a system call is needed.

Futex's

- A futex (short for "fast userspace mutex") is a basic tool to implement locking and building higher-level locking abstractions such as semaphores on Linux.

Futex's

- A futex consists of a piece of memory (an aligned integer) that can be shared among processes.
- It can be incremented and decremented by atomic assembler instructions, and processes can wait for the value to become positive.

Futex's

- Futex operations are done almost entirely in userspace.
- The kernel is only involved when a contended case requires arbitration.
- This allows locking primitives implemented using futexes to be very efficient.

Futex's

- Most operations do not require arbitration between processes.
- Most operations can be performed without needing to perform a (relatively expensive) system call.
- Only processor supplied atomic access/ modify instructions are used for userspace synchronization.